Efficient exact inference in discrete Anglican programs

Robert Cornish   Frank Wood   Hongseok Yang

University of Oxford

{rcornish,fwood}@robots.ox.ac.uk, hongseok.yang@cs.ox.ac.uk

1. Introduction

The design of a general probabilistic programming system involves a balancing act between two deeply conflicting concerns. On the one hand, the system should provide as uniform and flexible an interface for specifying models as possible; on the other, it should be capable of doing efficient inference for any particular model specified. Current systems lie somewhere on a spectrum that ranges from highly expressive languages such as Church [2], Anglican [10], and Venture [3], to highly performant languages like Figaro [7], FACTORIE [4], and Infer.NET [5]. It has not yet been shown possible to optimise both these concerns simultaneously.

To improve on this predicament we consider the class of discrete graphical models, for which various efficient exact inference algorithms exist. We present a technique for determining when an Anglican program expresses such a model, which allows us to achieve a substantial increase in inference performance in this case. Our approach can handle complicated language features including higher-order functions, bounded recursion, and data structures, which means that, for the discrete subset of Anglican, we do not incur any loss in expressiveness. Moreover, the resulting inference is exact, which can be useful in contexts where very high accuracy is required, or for doing nested inference inside larger models.

Details of Anglican can be found in [10], but for our purposes its semantics may be understood as the Clojure programming language augmented with a `sample` statement and an `observe` statement. `sample` takes as argument a distribution object Δ (which may be obtained by calling the built-in `flip` or `dirac` functions, for example) and produces a random sample from the corresponding mathematical distribution, which we denote P_Δ. On the other hand, `observe` takes as its arguments a distribution object Δ and a value v, and conditions the current execution trace on the observation $x_{\text{new}} = v$ for some new random variable $x_{\text{new}} \sim P_\Delta$.

In this work, we propose evaluating Anglican code according to a semantics that constructs a Bayes net on possible program values as a side effect of execution. Every intermediate value produced by evaluating some expression `M` is treated as a random variable, with a distribution that is conditional on the result of evaluating the subexpressions of `M`. We do not distinguish between stochastic and deterministic computation; for us, deterministic values simply have Dirac distributions conditioned on their inputs. Our semantics is non-standard in that we explore all execution traces that occur with some nonzero probability; in particular, evaluating an `if` statement usually involves evaluating both its branches. The output of our evaluator is a discrete graphical model that is then amenable to efficient exact inference via, for example, variable elimination [11]. An example is given in Program 1, which our method transforms into the model shown in Figure 1.

```
(defquery simple []
  (def y (sample (flip 0.5)))
  (def z (if y (dirac 5) (dirac 10)))
  (observe z 10)
  y)
```

Program 1: A simple program amenable to exact inference

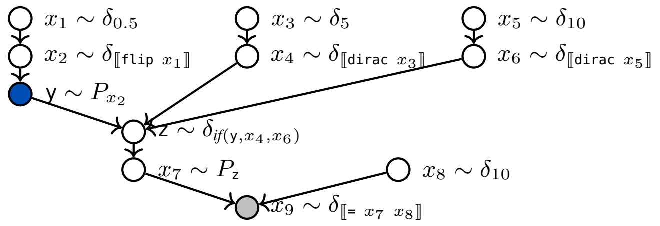
$x_1 \sim \delta_{0.5}$
$x_2 \sim \delta_{[\![\texttt{flip } x_1]\!]}$
$\mathsf{y} \sim P_{x_2}$
$x_3 \sim \delta_5$
$x_4 \sim \delta_{[\![\texttt{dirac } x_3]\!]}$
$x_5 \sim \delta_{10}$
$x_6 \sim \delta_{[\![\texttt{dirac } x_5]\!]}$
$\mathsf{z} \sim \delta_{\mathit{if}(\mathsf{y}, x_4, x_6)}$
$x_7 \sim P_{\mathsf{z}}$
$x_8 \sim \delta_{10}$
$x_9 \sim \delta_{[\![\texttt{=} \; x_7 \; x_8]\!]}$

Figure 1: Graphical model equivalent to Program 1. Here, δ_v denotes a Dirac distribution centered on v; $[\![\texttt{M}]\!]$ denotes the result of (deterministically) evaluating the expression `M` according to Clojure semantics; and *if* denotes a mathematical "if" function rather than a programmatic one. Each node corresponds to the value of a single expression that is evaluated in some execution path with nonzero probability, while edges correspond to immediate dependencies between these values. The nodes x_i correspond to intermediate values that are not assigned names in the program. The gray node is observed as having value `true`, and the blue node is our desired posterior.

2. Evaluator semantics

We now describe the semantics of our evaluator, which aims to convert Anglican code into an equivalent graphical model as in Figure 1. We will denote the *state* of execution by

$$\Sigma = \langle \gamma,\ \psi,\ \varphi,\ \omega,\ \eta,\ \alpha \rangle\,,$$

where: γ is a directed graph; ψ maps nodes to an over-approximation of their supports; φ maps nodes to their conditional distributions; ω is a set of observed nodes; η is the current *environment*, namely, a set of *frames* mapping variable names to their values; and α is the address of the currently active frame in η. The main components of interest are γ, ψ, φ, and ω, which constitute our desired graphical model; η and α are symbolic variants of standard devices used for implementing closures correctly, which we require in order to handle higher-order functions. We will adopt the convention of referring to the component γ of a state Σ_i directly as γ_i, and likewise for the other components.

Our treatment of literals, variables, lambdas, and definitions is straightforward, and our semantics for `sample` and `observe` statements can largely be understood from Figure 1. Big-step semantics for the most interesting Anglican expressions remaining are provided below. Our evaluation relation has the form $\langle \texttt{M},\ \Sigma_i \rangle \Downarrow \langle x_j,\ \Sigma_j \rangle$, where `M` is some expression, and x_j is some node in γ_j. Note also that each component of Σ_j should be assumed not to change from Σ_i unless otherwise stated.

2.1 Primitive applications

We evaluate applications of a first-order Clojure operator by first evaluating its arguments, and then applying the operator to all possible values that the arguments may take. That is, for f any first-order operator:

$$\frac{\langle \texttt{N}_i,\ \Sigma_{i-1} \rangle \Downarrow \langle x_i,\ \Sigma_i \rangle\,, 1 \le i \le k}{\langle f\ \ \texttt{N}_1\ \cdots\ \texttt{N}_k,\ \Sigma_0 \rangle \Downarrow \langle x_{\text{new}},\ \Sigma_{k+1} \rangle}\,,$$

where $x_{\text{new}} \notin \gamma_k$ is fresh, γ_{k+1} is γ_k plus an additional node for x_{new} with parents $x_1, \dots, x_k$, and

$$\begin{aligned} \varphi_{k+1} &= \varphi_k \left[\delta_{[\![f\ x_1\ \cdots\ x_k]\!]}\right] \\ \psi_{k+1} &= \psi_k \left[x_{\text{new}} \mapsto f\left(\prod_{1 \le i \le k} \psi_k(x_i) \right) \right]. \end{aligned}$$

2.2 Compound applications

Evaluating a compound application is somewhat complicated, since we must account for the fact that the value of its operator may be random, such as in Program 2.

```
(defn maybe-do-twice [f]
  (if (sample (flip 0.5)) (comp f f) f))
((maybe-do-twice inc) 0.5)
```

Program 2: A compound application whose operator is random

Our approach involves evaluating the application for all values in the support of the operator, and introducing a node to select from among these possibilities. Precisely:

$$\frac{\begin{gathered} \langle \mathtt{M},\ \Sigma_0 \rangle \Downarrow \langle x_1,\ \Sigma_1 \rangle \\ \langle \mathtt{N}_i,\ \Sigma_i \rangle \Downarrow \langle x_{i+1},\ \Sigma_{i+1} \rangle, 1 \le i \le k \\ \psi_1(x_1) = \{[\mathtt{fn}\ [\mathtt{y}_1 \cdots \mathtt{y}_k]\ \mathtt{P}_i,\ \alpha_i^*] \mid 1 \le i \le \ell\} \\ \langle \mathtt{P}_i,\ \overline{\Sigma}_{k+i} \rangle \Downarrow \langle x_{k+i+1},\ \Sigma_{k+i+1} \rangle, 1 \le i \le \ell \end{gathered}}{\langle \mathtt{M}\ \mathtt{N}_1\ \cdots\ \mathtt{N}_k,\ \Sigma_0 \rangle \Downarrow \langle x_{\text{new}},\ \Sigma_{k+\ell+2} \rangle},$$

where $x_{\text{new}} \notin \gamma_{k+\ell+1}$ is fresh, and $[\mathtt{fn}\ [\mathtt{y}_1 \cdots \mathtt{y}_k]\ \mathtt{P}_i,\ \alpha_i^*]$ denotes a closure. (Note that $\alpha_i^* \neq \alpha_i \in \Sigma_i$.) Further, for each $1 \le i \le \ell$ we have

$$\begin{aligned} \overline{\alpha}_{k+i} &= \alpha_{\text{new}} \\ \overline{\eta}_{k+i}(\alpha_{\text{new}}) &= \eta_{k+i}(\alpha_i^*)\,[\mathtt{y}_1 \mapsto x_2, \dots, \mathtt{y}_k \mapsto x_{k+1}], \end{aligned}$$

$\gamma_{k+\ell+2}$ and $\varphi_{k+\ell+2}$ are as shown in Figure 2, and

$$\begin{aligned} \alpha_{k+\ell+2} &= \alpha_0 \\ \psi_{k+\ell+2} &= \psi_{k+\ell+1} \left[x_{\text{new}} \mapsto \bigcup_{i=k+2}^{k+\ell+1} \psi_i(x_i) \right]. \end{aligned}$$

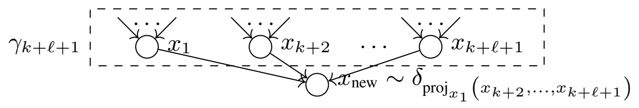
$\gamma_{k+\ell+1}$
x_1
x_{k+2}
$x_{k+\ell+1}$
$x_{\text{new}} \sim \delta_{\text{proj}_{x_1}(x_{k+2},\dots,x_{k+\ell+1})}$

Figure 2: $\gamma_{k+\ell+2}$ and $\varphi_{k+\ell+2}$; here $\text{proj}_{x_1}(x_{k+2}, \dots, x_{k+\ell+1})$ projects onto the x_i such that $x_1 = [\mathtt{fn}\ [\mathtt{y}_1, \dots, \mathtt{y}_n]\ \mathtt{P}_i,\ \alpha_i^*]$

2.3 If statements

In order to avoid *deterministic* recursion (such as in Program 3), we use multiple rules to evaluate `if` statements according to the support of its predicate. In particular, we use

$$\frac{\langle \mathtt{P},\ \Sigma_0 \rangle \Downarrow \langle x_1,\ \Sigma_1 \rangle \qquad \mathtt{false} \notin \psi_1(x_1) \\ \langle \mathtt{M},\ \Sigma_1 \rangle \Downarrow \langle x_2,\ \Sigma_2 \rangle}{\langle \mathtt{if\ P\ M\ N},\ \Sigma_0 \rangle \Downarrow \langle x_2,\ \Sigma_2 \rangle}$$

and the symmetric rule for when $\mathtt{true} \notin \psi_1(x_1)$. Finally, when both $\mathtt{true}, \mathtt{false} \in \psi_1(x_1)$, we proceed exactly as though `if` were a primitive operator and apply the rule in Section 2.1.

3. Discussion

Our approach may be understood as efficient enumeration-based inference, in which we propagate the supports of any random variables only to their immediate dependents. This entails a significant performance increase over the sort of naive enumeration technique referred to in [1], where execution is forked for each possible value of each random choice encountered at runtime. Whereas naive enumeration is exponential in the program's number of random choices, our evaluator requires time exponential only in the treewidth of our program, which is typically much smaller.

An interesting application of our approach is given by the Schelling coordination game [8], which involves two agents recursively reasoning about the other's preferences. Program 3 contains an Anglican version of the implementation found in [9]. Running this program in our evaluator and then doing variable elimination on the resulting Bayes net produces the exact conditional distribution of the value of `(bob depth)`, which existing inference methods within Anglican are only able to approximate. Moreover, our approach is fast: when `depth` $= 4$, we require $1.053\,\text{s}$ in total for evaluation and inference; when `depth` $= 8$, we require $6.47\,\text{s}$. In comparison, an implementation of the same program in the WebPPL probabilistic programming language required $5.46\,\text{s}$ and $9.10\,\text{s}$ for the same `depth` values when naive enumeration inference was used.

```
(defquery schelling-coordination-game [depth]
  (def location-dist (categorical {:good-bar 0.6
                                   :bad-bar 0.4}))
  (def alice (fn [depth]
               (let [alice-location (sample location-dist)]
                 (observe (dirac alice-location)
                          (bob (dec depth)))
                 alice-location)))
  (def bob (fn [depth]
             (let [bob-location (sample location-dist)]
               (if (> depth 0)
                 (observe (dirac bob-location) (alice depth)))
               bob-location)))
  (bob depth))
```

Program 3: Schelling Coordination Game

References

[1] N. D. Goodman. The Principles and Practice of Probabilistic Programming. *ACM SIGPLAN Notices*, 48(1):399–402, 2013.

[2] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. *CoRR*, abs/1206.3255, 2012.

[3] V. K. Mansinghka, D. Selsam, and Y. N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014.

[4] A. McCallum, K. Schultz, and S. Singh. FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In *Advances in Neural Information Processing Systems*, pages 1249–1257, 2009.

[5] T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer .NET 2.5. *Microsoft Research Cambridge*, 2012.

[6] A. Pfeffer. IBAL: A Probabilistic Rational Programming Language. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'01, pages 733–740, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[7] A. Pfeffer. Figaro: An Object-Oriented Probabilistic Programming Language. *Charles River Analytics Technical Report*, 137, 2009.

[8] T. C. Schelling. *The strategy of conflict*. Harvard university press, 1980.

[9] A. Stuhlmüller and N. D. Goodman. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research*, 28:80–99, 2014.

[10] F. Wood, J. W. van de Meent, and V. Mansinghka. A New Approach to Probabilistic Programming Inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

[11] N. L. Zhang and D. Poole. Exploiting Causal Independence in Bayesian Network Inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.